Automatic test purpose generation for Web services

Sébastien Salva

LIMOS CNRS UMR 6158
PRES Clermont University, Campus des Cézeaux
Aubière, FRANCE
sebastien.salva@u-clermont1.fr

Abstract. It is now well-established that to be reliable, software have to be tested during the software life cycle, and this is particularly true with recent technologies such as Web services. Test purpose based methods are black box testing techniques which take advantage of reducing the time required for test derivation. Nevertheless, test purposes must be constructed by hand. To solve this issue, we propose, in this paper, some automatic test purpose generation methods for testing the operation existence, the critical states and the exception handling, in stateful Web services. To take into account the SOAP environment in which they are deployed, we also augment the specification with SOAP messages. We show that SOAP gives more observable reactions and helps to test specific properties.

Keywords: Stateful Web services, STS, SOAP, test purpose generation

1 Introduction

Software testing is an important software engineering activity widely used to find defects in programs. In particular, black box testing, which is the topic of this paper, consists in testing a system implementation by means of test cases, usually constructed from a specification. This paper also focuses on Web services which represent interoperable components whose purpose is to externalize functional code in a standardized way, or the reuse of software accompanied by cost reduction.

Recently, several Web service based black box testing methods have been proposed [1–5]. Some of them are said exhaustive i.e. the test case selection is performed to ensure that a faulty implementation is detected by a least one test case. Nevertheless, this exhaustiveness often implies a costly test case generation which eventually may lead to a state space explosion. Moreover, the test case set is not exhaustive in practice: service oriented application specifications are often symbolic which means that these latter are composed of variables and guards. The variable domain is often infinite and impossible to test completely.

Test purpose based methods represent an interesting alternative. Test purposes are test requirements which are given by designers. They can be used to test various properties such as the critical states, the coverage of specific actions, etc. The test selection is then guided and thereby reduced since test purposes aim to target the test of some implementation parts only. Some works dealing with test purpose based methods for Web services have been proposed recently [3–5]. These methods generate test cases by synchronizing test purposes with the specification to produce action sequences which respect the specification and which contain the test purpose properties. Then, test cases are experimented on the implementation under test to conclude whether test purposes are satisfied.

Although using this approach greatly reduces test costs, the main encountered issue is that test purposes are formulated manually. And, constructing them is particularly difficult when the system is large, has real-time constraints or is distributed. However, many test purposes can be generated automatically as it has been showed in some works [6] which propose test purpose generation techniques for specific untimed systems (distributed systems and protocols). But to our knowledge, none method has been proposed for service oriented applications. This is why we present, in this paper, several techniques to generate test purposes for SOAP Web services, modelled with Symbolic Transition Systems (STS [7]). Usually, Web services are deployed in specific environments, e.g., HTTP for REST Web services or SOAP [8]. We show that the latter modifies the behaviour of the tested Web services and may give new relevant information (specific messages) for testing. So, the originality of our approach is to augment the specification to take into account the SOAP environment in order to test specific properties e.g., the exception handling. From the completed specification, we propose new test purpose generation methods to test the operation existence, the critical states and the exception handling.

This paper is structured as follows: Section 2 defines the specification and test purpose modelling. We describe the advantages granted by SOAP for testing in section 3 and define the specification completion. Test purpose generation methods are given in section 4. We provide some experiment results in section 5. And finally, section 6 gives some perspectives and conclusions.

2 Web service and test purpose modelling

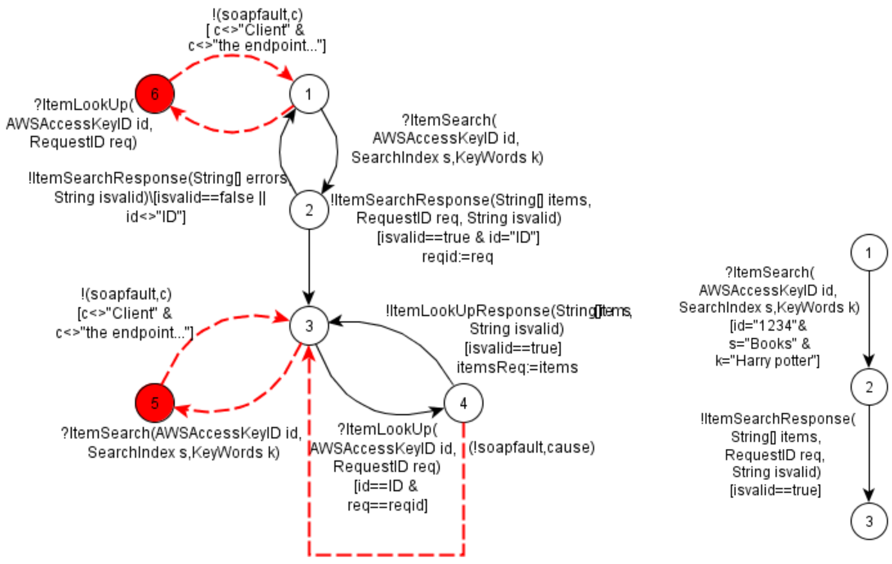
We formalize, in this paper, Web services with Symbolic Transition Systems (STS [7]). This extended automaton model associates a behaviour with a specification composed of transitions labelled by actions and of internal and external variables sets, which may be used to send or receive concrete values and to set guards which must be satisfied to fire transitions. Below, we only summarize the suspension STS definition where quiescence (the lack of observation) is taken into account with the δ symbol. The complete definition can be found in [7].

Definition 1. *A (suspension) Symbolic Transition System STS is a tuple* $< L, l_0, V, V_0, I, \Lambda, \rightarrow >$*, where:*

- *L is the finite set of locations, with l_0 the initial one,*
- *V is the finite set of internal variables, I is the finite set of external or interaction ones. We denote D_v the domain in which a variable v takes*

values. The internal variables are initialized with the assignment V_0, which is assumed to take an unique value in D_V,

- *Λ is the finite set of actions, partitioned by $\Lambda = \Lambda^I U \Lambda^O$: inputs, beginning with ?, are provided to the system, while outputs (beginning with !) are observed from it. $a(p) \in \Lambda$ is an action where $p = (p_1, ..., p_k)$ is a finite set of external variables. We denote type$(p) = (t_1, ..., t_k)$ the type of the variable set p. δ denotes the quiesence i.e. the lack of observation from a location,*
- *$\rightarrow$ is the finite transition set. A transition $(l_i, l_j, a(p), \varphi, \varrho)$, from the location $l_i \in L$ to $l_j \in L$, also denoted $l_i \xrightarrow{a(p),\varphi,\varrho} l_j$ is labelled by $a(p) \in \Lambda$, $\varphi \subseteq D_V \times D_p$ is a guard which restricts the firing of the transition. Internal variables are updated with the assignment $\varrho : D_V \times D_p \rightarrow D_V$ once the transition is fired.*



(a) A Web service specification (b) A test purpose

Fig. 1.

The STS model is not specifically dedicated to Web services. These latter may be invoked with methods called operations. This is why, for modelling, we assume that an action $a(p)$ in Λ represents either the invocation of an operation *op* which is denoted *opReq* or the return of an operation *op* with *opResp*. For an STS $\mathcal{S}$, we denote $\mathcal{OP}(\mathcal{S})$ the operation set found in Λ. A specification example, is illustrated in Figure 1(a) (black transitions). This one describes a part of the Amazon Web Service devoted for e-commerce (AWSECommerceService [9]). For sake of simplicity, we only consider two operations "ItemSearch", which aims to search for items, and "ItemLookUp", which provides more details about an item. Note that we do not include all the parameters for readability reasons.

On the other hand, test purposes describe the test intention. We assume that these ones are composed exclusively of specification properties which should be met in the implementation under test. Usually, test purposes do not represent complete specification paths. Therefore, they are often synchronized with the specification to generate executable test cases. Consequently, we also formalize a test purpose with a deterministic and acyclic STS $TP =< L_{TP}, l0_{TP}, V_{TP}, V0_{TP}, I_{TP}, \Lambda_{TP}, \rightarrow_{TP}>$ such that $\rightarrow_{TP}$ is composed of transitions modelling specification properties. So, for any transition $l_j \xrightarrow{a(p),\varphi_j,\varrho_j} l'_j \in\rightarrow_{TP}$, it exists a transition $l_i \xrightarrow{a(p),\varphi_i,\varrho_i} l'_i \in\rightarrow$ and a value set $(x_1, ..., x_n) \in D^n_{V\cup I}$ such that $\varphi_j \wedge \varphi_i(x_1, ..., x_n) \models$ true. A test purpose example is illustrated in Figure 1(b). This one aims to search for books whose description contain the keywords "Harry potter". We must obtain a valid response.

3 The advantages offered by the SOAP environment for testing

Web services are deployed in specific environments, e.g., SOAP for SOAP Web services, to structure messages in an interoperable manner and to manage operation invocations. In particular, the SOAP environment consists in a SOAP layer which serializes messages with XML and of SOAP receivers (SOAP processor + Web services) [10] which is software, in Web servers, that consumes messages. The SOAP processor is a Web service framework part which represents an intermediary between client applications and Web services and which serializes/deserializes data and calls the corresponding operations. The significant modifications involved by SOAP processors can be found in [11].

In summary, SOAP processors add new messages, called SOAP faults, which give details about faults raised in the server side. They return SOAP faults composed of the causes "Client" or "the endpoint reference not found" if services or operations or parameter types do not exit. SOAP processors also generate SOAP faults when a service instance has crashed while triggering exceptions. In this case, the fault cause is equal to the exception name. However, exceptions correctly managed in the specification and in the service code (with try...catch blocks) are distinguished from the previous ones since a correct exception handling produces SOAP faults composed of the cause "SOAPFaultException". So, SOAP faults can also be used to test whether the exception handling is correct by identifying the received causes. Consequently, taking into consideration these messages while generating test purposes sounds very interesting to check the satisfaction of specific properties e.g, the exception handling. So, we propose to augment the specification with the SOAP faults generated by SOAP processors. We denote $(soapfault, cause)$ a SOAP fault where the variable $cause$ is the reason of the SOAP fault receipt.

Let $S =< L, l_0, V, V_0, I, \Lambda, \rightarrow>$ be a Web service specification. S is completed by means of the STS operation *addsoap* in S which augments the specification with SOAP faults as described previously. The result is an STS $S\uparrow$. The operation *addsoap* is defined as follow: *addsoap* in $S =_{def} S\uparrow =< L_{S\uparrow}, l_0, V, V_0, I,$

$\Lambda_{S\uparrow}, \rightarrow_{S\uparrow}>$ where $L_{S\uparrow}$, $\Lambda_{S\uparrow}$ and $\rightarrow_{S\uparrow}$ are defined by the following inference rules:

$$R_1 : \frac{l_1 \xrightarrow{?opReq(p),\varphi,\varrho} l_2 \in \rightarrow_S, l_1 \xrightarrow{?op'Req(p),\varphi',\varrho'} l \notin \rightarrow_S,}{l \xrightarrow{?op'Req(p),\emptyset,\emptyset} l' \in \rightarrow_{S\uparrow}, l' \xrightarrow{!a(p),\varphi,\emptyset} l \in \rightarrow_{S\uparrow}, \varphi = [a(p) \neq (soapfault, "CLIENT") \wedge \frac{l' \notin L_S}{a(p) \neq (soapfault, \text{"the endpoint reference not found"})]}}$$

$$R_2 : \frac{l \xrightarrow{?opReq(p),\varphi,\varrho} l' \in \rightarrow_S, \varphi' = \bigwedge_{l' \xrightarrow{!opResp_i(r_i),\varphi_i,\varrho_i} l'_i \in \rightarrow_S} \neg\varphi_i}{l' \xrightarrow{!(soapfault,cause),\varphi',\emptyset} l}$$

The first rule completes the initial specification on the input set by assuming that each unspecified operation request returns a SOAP fault message. The second rule completes the output set by adding, after each transition modelling an operation request, a transition labelled by a SOAP fault. Its guard corresponds to the negation of the guards of transitions modelling responses. A completed specification is illustrated in Figure 1(a) with dashed transitions.

4 Automatic Test Purpose generation methods

Although test purposes sound interesting to reduce test costs, these ones also raise an important drawback since they are usually formulated manually. So, we contribute to solve this issue by introducing some automatic generation techniques for Web services. We assume having a completed specification $S\uparrow$. We propose three test purpose generation approaches which aim to test the operation existence, the critical locations, and the exception handling.

Operation existence testing

This approach generates test purposes for testing whether operations in $\mathcal{OP}(S\uparrow)$, with $S\uparrow$ an STS specification, are implemented and can be invoked. With the specification completion, detailed in the previous section, it becomes possible to test the existence of any operation, even those which do not return any response, i.e. any observable reaction. Indeed, if an operation is not implemented as it is described in the specification, the SOAP processor will return a SOAP fault composed either of the cause "Client" or of the cause "the end point reference not found". So, for a specification $S\uparrow =< L_{S\uparrow}, l0_{S\uparrow}, V_{S\uparrow}, V0_{S\uparrow}, I_{S\uparrow}, \Lambda_{S\uparrow}, \rightarrow_{S\uparrow}>$, the test purpose set is given by:

$$TP = \bigwedge_{op \in \mathcal{OP}(S\uparrow)} \{tp =< L, l_0, V_S, V0_S, I_S, \Lambda, \rightarrow> \text{ where } \rightarrow = \{l_0 \xrightarrow{?opReq(p),\emptyset,\emptyset} l_1, l_1 \xrightarrow{!a(p),\varphi,\emptyset} l_2, \text{ with } \varphi = [a(p) \neq (soapfault, "Client") \wedge a(p) \neq (soapfault, \text{"the end point reference not found"})]\}\}$$

The specification of Figure 1(a) is composed of two operations, so we obtain two test purposes. These ones will be synchronized later with the specification to test any operation invocation.

Critical location testing

The second technique aims at testing the specification critical locations. It is not obvious to set which location is critical since no general and formal definition

is given in literature. So, in this paper, we suggest that the critical locations are those the most potentially encountered in the acyclic specification paths. Nevertheless, other criteria could be chosen, such as the less visited locations, or the quiescent ones. We give in [11] an algorithm which is derived from the DFS (Depth First Search) one, to detect the critical location set, denoted CS. Then, for each critical location $l \in CS$, we construct test purposes to test all the outgoing transitions of l. The test purpose set, expressed below, is composed of specification paths finished by output actions to observe the implementation reactions while testing. For a specification $S\uparrow =< L_{S\uparrow}, l0_{S\uparrow}, V_{S\uparrow}, V0_{S\uparrow}, I_{S\uparrow}, \Lambda_{S\uparrow}, \rightarrow_{S\uparrow}>$, the test purpose set is given by:
$TP = \bigwedge_{l \in CS} \{tp =< L, l_0, V_S, V0_S, I_S, \Lambda, \rightarrow>$ where $\rightarrow$ is constructed with the following inferences rules:

$$R_1 : \frac{l \xrightarrow{!a(p),\varrho,\varphi} l' \in \rightarrow_{S\uparrow}, a(p) \neq \delta}{l_0 \xrightarrow{!a(p),\varrho,\varphi} l' \in \rightarrow}$$

$$R_2 : \frac{l \xrightarrow{?a(p),\varrho,\varphi} l' \in \rightarrow_{S\uparrow}, p = l' \xrightarrow{a_1(p),\varrho_1,\varphi_1} l'_1 \ldots l'_{n-1} \xrightarrow{a_n(p),\varrho_n,\varphi_n} l'_n \in (\rightarrow_{S\uparrow})^n, a_n(p) \in \Lambda^O_{S\uparrow}/\{\delta\}}{l_0 \xrightarrow{!a(p),\varrho,\varphi} l'.p \in (\rightarrow)^{n+1}}$$

R_1 is used when an outgoing transition, from a critical location, is labelled by an output. In this case, this transition is added to the test purpose. The second rule is used when a transition is labelled by an input. The test purpose is completed with this transition followed by a specification path finished by an output. A test purpose generation algorithm is given in [11]. In the specification of figure 1(a) we have two critical locations l_2 and l_3. So, we obtain two test purposes which aim to test all the outgoing transitions of l_2 and l_3 with paths finished by output actions.

Exception handling testing

As described in Section 3, SOAP processors return SOAP faults when exception are triggered in a Web service operation at runtime. SOAP processors also enable to differentiate the exceptions resulting of unexpected Web service crashes from those which are thrown in Web service operations (with try ... catch blocks for instance). In the last case only, we obtain SOAP faults composed of the "SoapFaultException" cause.

With the specification completion described in section 3, we can construct test purposes to test whether the exception handling is correctly implemented and not managed by SOAP processors. However, to trigger exceptions, test purposes must be formulated over predefined value sets, that we denote $U(t)$. These ones are composed of unusual values well known for relieving bugs, for any simple or complex type t. For instance, $U(string)$ is composed of the values &", "$", null or "_", which usually trigger exceptions. For a specification $S\uparrow =< L_{S\uparrow}, l0_{S\uparrow}, V_{S\uparrow}, V0_{S\uparrow}, I_{S\uparrow}, \Lambda_{S\uparrow}, \rightarrow_{S\uparrow}>$, the test purpose set is given by:
$TP = \bigwedge_{l \xrightarrow{?opReq(p),\varphi,\varrho} l' \in \rightarrow_{S\uparrow}} \{tp =< L, l_0, V_{S\uparrow}, V0_{S\uparrow}, I_{S\uparrow}, \Lambda, \rightarrow>$ where $\rightarrow =$

$\{l_0 \xrightarrow{?opReq(p),\varphi',\varrho} l_1, l_1 \xrightarrow{(!soapfault,"SOAPFaultException"),\emptyset,\emptyset} l_2$ where $\varphi' = \varphi \wedge p = (p_1, ..., p_n)$ takes values in $U(type(p_1)) \times ... \times U(type(p_n))\}$

The specification of Figure 1(a) contains four operation requests from locations l_1 and l_3. If we suppose that $card(U(type(p_1)) \times ... \times U(type(p_n))) = n$, we obtain at most $4n$ test purposes. It is manifest that the larger the unusual values sets, the larger the test purpose set will be. To limit it, instead of using a cartesian product, other solutions may be used such as pairwise testing [12] which constructs discrete combinations for pair of parameters only and which has been shown sufficient to cover parameter domains.

5 Experimentation

At the moment, we have implemented a preliminary tool which performs the test purpose generation from a completed STS and the synchronous products between the specification and test purposes. Then, we have manually extracted test cases and translated them into the Soapui format. Then, these ones can be executed with the Soapui tool [13] which aims to experiment Web services with unit test cases. A Soapui test case example can be found in an extended version of this paper in [11]. We applied the test purpose generation on the AWSECommerceService (09/10 version). Results are given in Figure 2. All the 22 operations handle a large number of parameters, therefore we limited the test purpose number to 10 per operation, for the exception handling method. We obtained fail verdicts only for the exception handling tests. Indeed, we obtained some SOAP faults composed of the cause *Client*, meaning that the requests are incoherent although the test cases satisfy the specification. We also received unspecified messages corresponding to errors composed of a wrong cause. For instance, instead of receiving SOAP faults, we obtained the response "Your request should have at least 1 of the following parameters: AWSAccessKeyId, SubscriptionId when we called the operation CartAdd with a quantity equal to "-1", or when we searched for a "Book" type instead of the "book" one, whereas the two parameters AWSAccessKeyId, SubscriptionId were right.

	Existence	Critical locations	Exception handling
test purposes	22	2	22
test cases	44	22	210
fail verdicts	0	0	39

Fig. 2. Test results on the Amazon AWSECommerceService Service

6 Conclusion

We have proposed, in this paper, some methods to generate automatically test purposes from a Stateful Web service specification. We believe that these latter are relevant when used in combination with existing test purpose based methods to produce test cases automatically and to prevent from writing test purposes manually.

We have also shown that taking into account the SOAP environment during the test brings new information which help to test specific properties such as the operation existence or the exception handling. An immediate line of future work is to propose other generation approaches such as the test of the location accessibility. We also intend to extend this work on service compositions to test composition properties.

References

1. García-Fanjul, J., Tuya, J., de la Riva, C.: Generating test cases specifications for compositions of web services. In Bertolino, A., Polini, A., eds.: in Proceedings of International Workshop on Web Services Modeling and Testing (WS-MaTe2006), Palermo, Sicily, ITALY (2006) 83–94
2. Frantzen, L., Tretmans, J., de Vries, R.: Towards model-based testing of web services. In Bertolino, A., Polini, A., eds.: in Proceedings of International Workshop on Web Services Modeling and Testing (WS-MaTe2006), Palermo, Sicily, ITALY (2006) 67–82
3. Lallali, M., Zaidi, F., Cavalli, A., Hwang, I.: Automatic timed test case generation for web services composition. In Press, I.C.S., ed.: The 6th IEEE European Conference on Web Services (ECOWS'08), Dublin (2008) 53–63.
4. Escobedo, J.P., Gaston, C., Gall, P., Cavalli, A.: Observability and controllability issues in conformance testing of web service compositions. In: TESTCOM '09/FATES '09, Berlin, Heidelberg, Springer-Verlag (2009) 217–222
5. Cao, T.D., Felix, P., Castanet, R.: Wsotf: An automatic testing tool for web services composition. In: Proceedings of the 2010 Fifth International Conference on Internet and Web Applications and Services. ICIW '10, Washington, DC, USA, IEEE Computer Society (2010) 7–12
6. Henniger, O., Lu, M., Ural, H.: Automatic generation of test purposes for testing distributed systems. In Petrenko, A., Ulrich, A., eds.: FATES. Volume 2931 of Lecture Notes in Computer Science., Springer (2003) 178–191
7. Frantzen, L., Tretmans, J., Willemse, T.: Test Generation Based on Symbolic Specifications. In Grabowski, J., Nielsen, B., eds.: Formal Approaches to Software Testing – FATES 2004. Number 3395 in Lecture Notes in Computer Science, Springer (2005) 1–15
8. Consortium, W.W.W.: Simple object access protocol v1.2 (soap). (2003)
9. Amazon: Amazon e-commerce service. (2010) http://docs.amazonwebservices.com/AWSEcommerceService/4-0/.
10. organization, W.I.: Ws-i basic profile. (2006) http://www.ws-i.org/docs/charters/WSBasic_Profile_Charter2-1.pdf.
11. Salva, S., Rabhi, I.: Automatic test purpose generation for Web services. (2011) LIMOS Research report RR-11-04.
12. Cohen, M.B., Gibbons, P.B., Mugridge, W.B.: Constructing test suites for interaction testing. In: Proc. Intl. Conf. on Software Engineering (ICSE). (2003) 38–48
13. Eviware: Soapui. (2011) http://www.soapui.org/.